# Programming Level

A.R. Hurson
Department of Computer Science
Missouri University of Science & Technology
Rolla, Missouri 65409
hurson@mst.edu

# Module2 Background

- **Programming Level**
  - **Computer:** A computer with a storage component that may contain both data to be manipulated and instructions to manipulate the data is called a **stored program machine**. This simply implies that the user is able to change the sequence of operations on the data.
  - **Program:** The sequence of operations performed on the data is called a **program**. More formally, it is a finite set of instructions that specify the operands, operations, and the sequence by which processing has to occur.

# Module2 Background

- **Programming Level**
  - **Instruction:** An instruction is a group of bits that tells the computer to perform a specific operation. It is composed of two parts:
    - **Operation part**
    - **Operand part**

# Module2 Background

- **Programming Level**
  - **Operation part** (operation code) is a group of bits that defines the action to be performed.
    - For each machine the set of operations is limited and defined.
    - In general, a machine with *n* bits as op. code is capable of supporting $2^n$ distinct operations each having a distinct encoding as op. code.

# Module2 Background

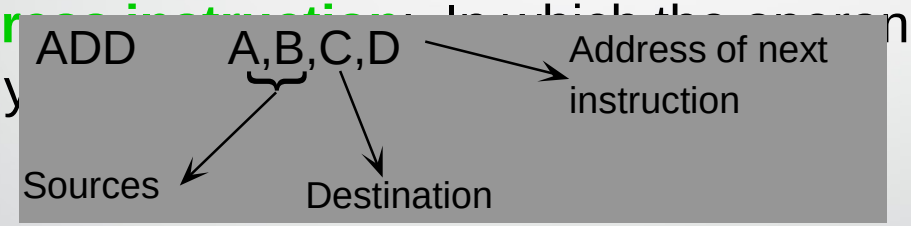
- **Programming Level**
  - **Operand part** defines the element (s) needed for operation.
  - Within the scope of a program, besides the op. code, one needs four pieces of information (note majority of operations are binary operations):
    - 2 operands as sources
    - 1 operand as a destination
    - 1 operand to specify the location of the next instruction that should be fetched.

# Module2 Background

- **Programming Level**
  - Depending on how many of these pieces of information are explicitly defined, instructions are grouped into 5 classes: 4, 3, 2, 1, and 0-address instructions.

# Module2 Background

- **Programming Level**
  - **4 - addr** nd part explicitly




  - Meaning: add contents of A to B and store the result in C and then fetch the next instruction from D.

# Module2 Background

- **Programming Level**
  - **3 - address instruction**: Most of the instructions in a program are sequential in nature. Therefore, it is possible to eliminate the address of the next instruction from the operand part and assume that the next instruction is in the next consecutive location in main memory. This brings the concept of 3-address instruction.

# Module2 Background

- **Programming Level**
  - In case of **3 - address instruction** a register, program counter (PC), is used as a pointer to point to the next instruction in sequence.

    i) ADD A,B,C
  - Meaning: add the contents of A to B and store the result in C. In this case the program counter (PC) contains (i + 1).

# Module2 Background

- **Programming Level**
  - **2 - address instruction**: If the result of the operation is going to be stored in one of the sources then we will end up with a so-called 2-address instruction.

    i) ADD A,B
  - Meaning: add contents of A to B and store it either in A or B. In this case the PC contains (i+1).

# Module2 Background

- **Programming Level**
  - **1 - address instruction**: In a 2-address instruction, if one of the operands is being implicitly defined, then only one source is needed to be explicitly defined in the operand part. This brings the concept of 1-address instruction. This has been implemented by the introduction of a register, accumulator (AC), which acts as a source as well as destination.

    i) ADD A
  - Meaning: add the contents of A to the AC and store the result in the AC, PC contains (i + 1).

# Module2 Background

- **Programming Level**
  - **0 - address instruction**: Both operands are implicitly defined and hence just the op.code is explicitly defined in the instruction. This implementation is the so-called stack machine, where operand values are assumed to be on top of the stack.

  i) ADD

  - Meaning: pop stack twice, add the contents of the two top most elements together and push the result back into the stack. PC contains (i + 1).

# Module2 Background

- **Programming Level**
  - Calculate the length of 4, 3, 2, 1, and 0-address instructions for a machine capable of supporting 15 instructions and a main memory of 16K:
    - 4-address instruction: 4 + 4 * 14 = 60
    - 3-address instruction: 4 + 3 * 14 = 46
    - 2-address instruction: 4 + 2 * 14 = 32
    - 1-address instruction: 4 + 1 * 14 = 18
    - 0-address instruction: 4

# Module2 Background

- **Programming Level**
  - In case of 2-address instructions, the destination operand loses its initial value. So in many cases it is advisable to save its initial value: i.e.,

    MOVE A to T (T is a temporary area)

  - In case of 1-address instructions, the accumulator should be initialized before performing operations. Two instructions, namely **LOAD** and **STORE** are used to move data to and from accumulator:

    LOAD A,

    STORE A

# Module2 Background

- **Programming Level**
  - In case of 0-address instructions, stack should be loaded first. Two instructions namely **PUSH** and **POP** are used to move data between main memory and stack:

```
PUSH A
POP  A
```

# Module2 Background

- **Questions**
  - Compare and contrast 4, 3, 2, 1, and 0-address instructions and programs against each other.
  - Is it possible to write a pure 0-address program?
  - What determines the length of PC?

# Module2 Background

- Working Problem
  - Write a 4, 3, 2, 1, and 0-address program for

    Y = A ** B - (C + D)

    Also calculate the program size based on the physical configuration of the system we discussed before.

# Module2 Background

- Working Problem
  - **4-address program**

  1) EXP A,B,$T_1$,2
  2) ADD C,D,$T_2$,3
  3) SUB $T_1$,$T_2$,Y,4
  4) ...

  Instruction length = 60 bits

  Program size = 3 * 60 = 180 bits

# Module2 Background

- Working Problem
  - **3 - Address Program**

  1) EXP A,B,$T_1$ PC=2
  2) ADD C,D,$T_2$ PC=3
  3) SUB $T_1$,$T_2$,Y PC=4
  4) ...

  Instruction length = 46 bits

  Program size = 3 * 46 = 138 bits

# Module2 Background

- Working Problem
  - **2 - Address Program**

    **1) MOVE A, $T_1$ PC=2**

    **2) EXP $T_1$, B PC=3**

    **3) MOVE C, $T_2$ PC=4**

    **4) ADD $T_2$, D PC=5**

    **5) SUB $T_1$, $T_2$ PC=6**

    **6) MOVE $T_1$, Y PC=7**

    **7)...**

    **Instruction length = 32 bits**

    **Program size = 6 * 32 = 192 bits**

# Module2 Background

- **Programming Level**
  - **1 - Address Program**

1) LOAD A PC=2
2) EXP B PC=3
3) STORE $T_1$ PC=4
4) LOAD C PC=5
5) ADD D PC=6
6) STORE $T_2$ PC=7
7) LOAD $T_1$ PC=8
8) SUB $T_2$ PC=9
9) STORE Y PC=10
10) ...

Instruction length = 18 bits

Program size = 9 * 18 = 162 bits

# Module2 Background

- **Programming Level**
  - **0 - Address Program**

```
1)  PUSH   A        PC=2
2)  PUSH   B        PC=3
3)  EXP             PC=4
4)  PUSH   C        PC=5
5)  PUSH   D        PC=6
6)  ADD             PC=7
7)  SUB             PC=8
8)  POP    Y        PC=9
```

| Instruction length | 0 - Address 4 bits<br>1 - Address 18 bits |
|---|---|

Program size = 18 * 5 + 3 * 4 = 102 bits

# Module2 Background

- Reading Assignment
  - Chapter 2 (sections 2.1-2.9)
- Homework2
  - Due September 22

# Module2 Background

- **Programming Level**
  - The order of the instructions in a program should be the same as the order of instructions in a post-fix format – Expression should be converted into a post-fix format.
  - In-fix to post-fix conversion:
    - Make fully parenthesized expression
    - Move each operator to its nearest right parenthesis
    - Delete all the left and right parentheses.

# Module2 Background

- **Prog**
  - Co

$$A + B \equiv (A + B) \equiv (\overset{+}{A}B) \equiv A\,B\,+$$

$$A/(B + C) * D \equiv ((A/(B + C)) * D) \equiv ((A(BC)\overset{+/}{)}\overset{*}{D}) \equiv ABC\,+/\;D\,*$$

$$(A * X + B) / (C * X - D) \equiv (((A * X) + B)/((C * X) - D)) \equiv$$

$$(((A\overset{*}{X})\overset{+}{B})((C\overset{*}{X})\overset{-/}{D})) \equiv AX * B + C\,X * D\,-\,/$$

# Module2 Background

- **Addressing Mode**
  - The way in which operands are specified in an instruction is called the **addressing mode**. The ability to specify operands in different ways brings a greater degree of flexibility to the computer.
    - **Implied Mode**: In this mode operand(s) is(are) specified implicitly in the definition of instruction.

      ADD A  Accumulator is in implied mode.
    - **Immediate Mode**: In this mode operand value is defined in the instruction.

      ADD 5

# Module2 Background

- **Addressing Mode**
  - **Index Mode**: In this mode address of the operand is determined by two values:
    - Contents of the index register
    - Displacement
    - **Effective address** - i.e., address of operand, is determined by adding the contents of index register to the displacement. Index mode is very effective when handling arrays, tables, …
  - **Register Mode**: In this mode operand value is contained in a register which is referenced to by the instruction. Usually, such a mode is used in the system with multiple registers.

ADD $R_1$ To $R_2$

# Module2 Background

- **Addressing Mode**
  - **Direct Addressing Mode**: In this mode the address of operand is explicitly defined in the instruction.

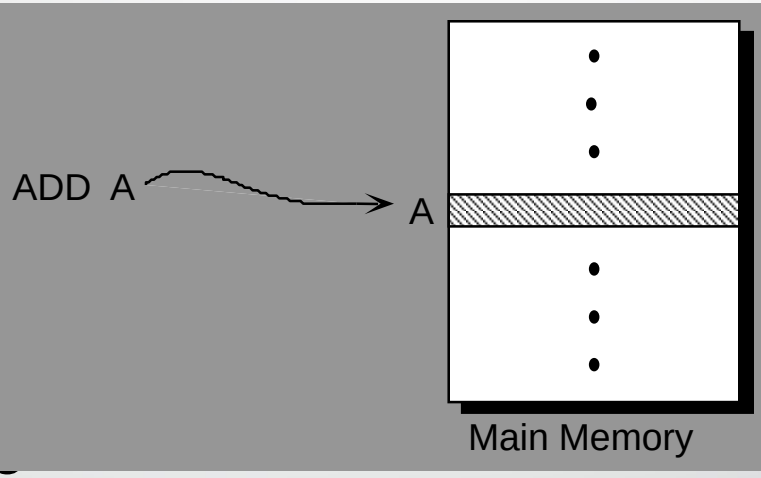
  ADD A TO B

  Register mode and direct mode are conceptually the same.
  - **Indirect Mode**: In this mode, the address of the address of operand value is contained in the instruction. Therefore, one needs two fetches to memory in order to retrieve the operand value.

  ADD $A_{indirect}$ TO B
  - **Multilevel of Indirections**: This is an extension of indirect mode, in which instruction contains the address of the address of the ... of operand.

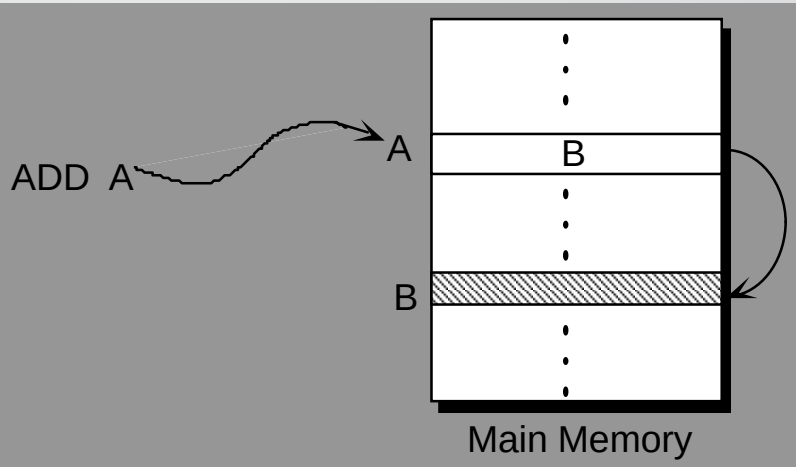
# Module2 Background

**Direct Mode:**

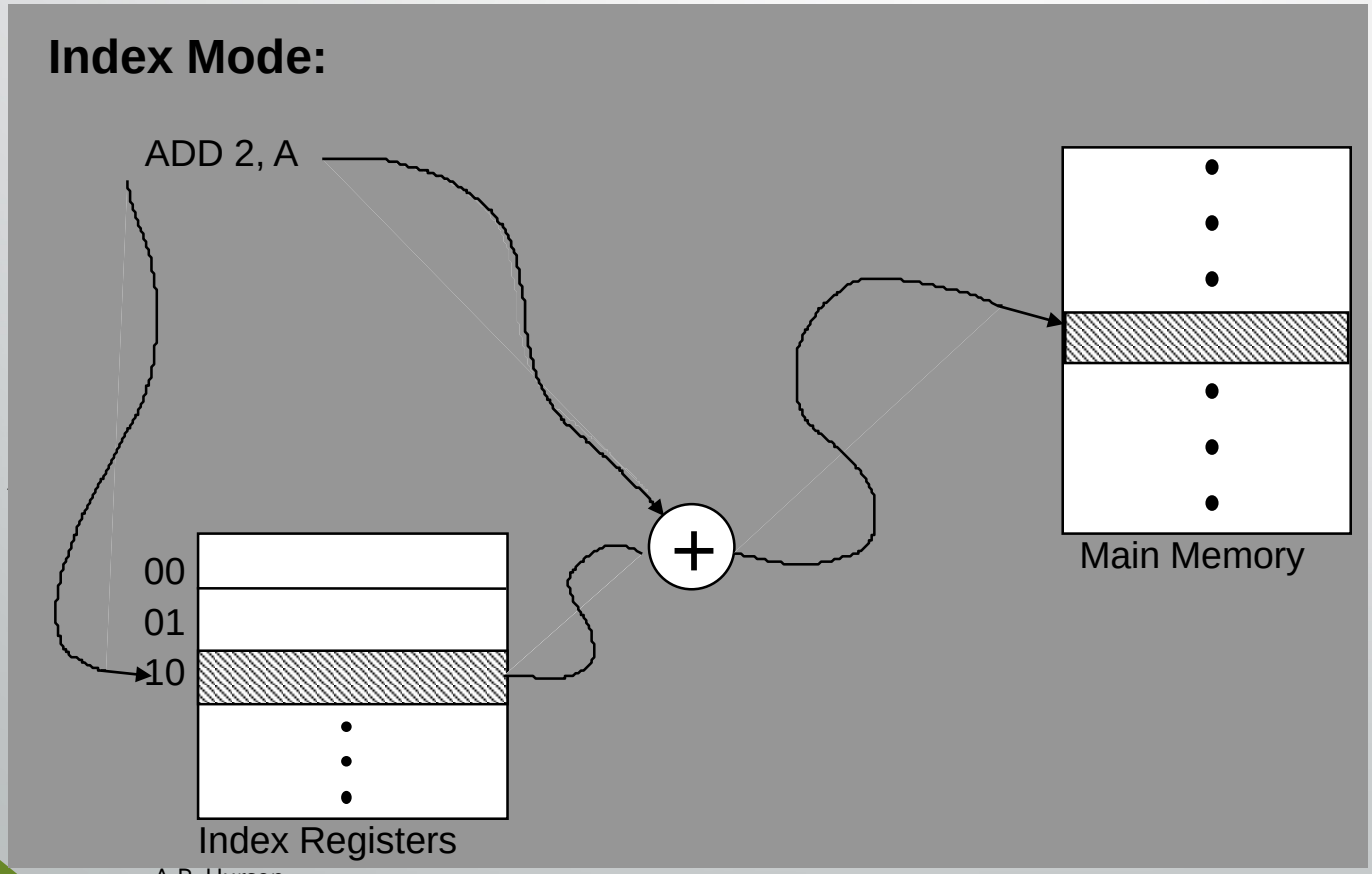



- **Addressing Mode**

**Indirect Mode:**



# Module2 Background



# Module2 Background

- **Addressing Mode**
  - A variation of index mode is called **base addressing** which allows re-locatability.
  - Some systems allow a mixture of several addressing modes, i.e., **indirect-register mode**.

# Module2 Background

- **Questions**
  - How are different addressing modes defined in an instruction?
  - How is the level of indirection defined?
  - What is the application of multi-level of indirection?
  - If 5 and 6 are immediate values then are

  ADD 5 to 6

  ADD 5 to A

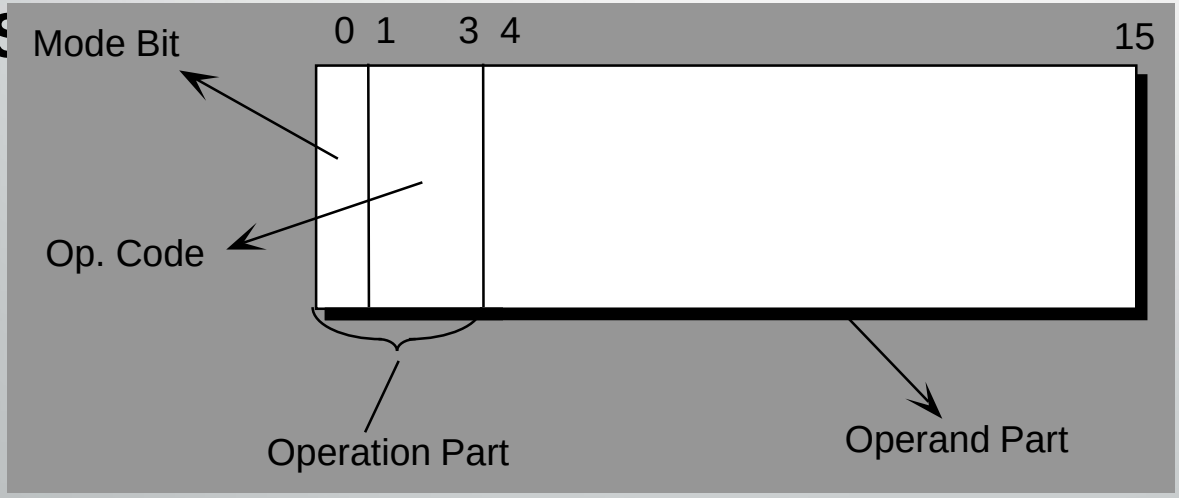
  valid instructions?
  - What is the format and length of a 2-address instruction, where:
    instruction set is of size 15
    memory is of size 16K words
    system supports 3 different addressing modes (immediate, direct, and indirect).

# Module2 Background

- **Study of a Simple Machine**
  - The simplicity of this machine is due to the simplicity of its control unit, the limited number of operations it supports, and the simplicity of its instruction format.
  - General Configuration
    - Simple machine is a binary, $2^s$ complement, 1-address machine.
    - Main memory is of size 4096 * 16
    - MAR is of length 12 bits
    - MBR is of length 16 bits
    - PC is of length 12 bits
    - AC is of length 16 bits
    - AC is extended by 1 bit register called E-bit (extended bit).
    - Instruction register is composed of two parts:
      - OPR 3-bit register
      - I 1-bit register

# Module2 Background

- S

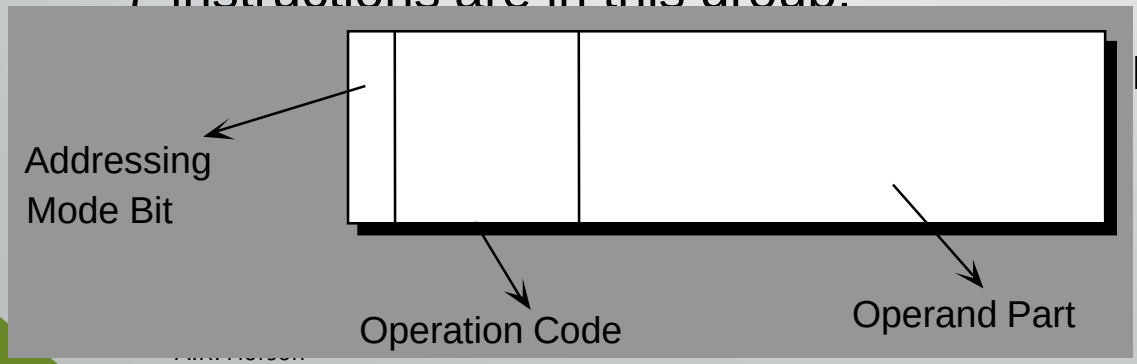


# Module2 Background

- **Study of a Simple Machine**
  - The simple machine supports 2 types of addressing modes:
    - Direct (0)
    - Indirect (1)
  - Instruction Formats: Instruction set is partitioned into three groups
    - **Memory reference instructions**
    - **Register reference instructions**
    - **Input-Output instructions**

# Module2 Background

- **Study of a Simple Machine**
  - **Memory reference instructions**: These instructions explicitly refer to a location in memory as the operand.
  - 7-instructions are in this group:



# Module2 Background

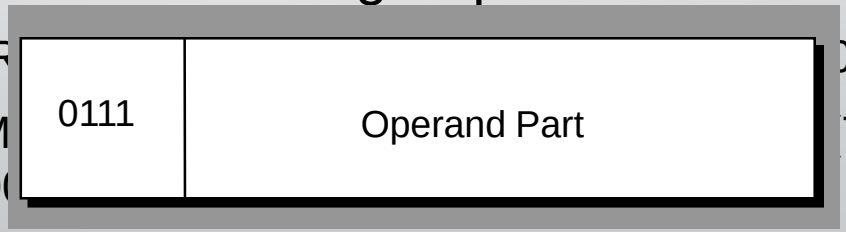
| Symbol | Code | Description |
|---|---|---|
| AND | 000 | And |
| ADD | 001 | Add |
| LDA | 010 | Load |
| STA | 011 | Store |
| BUN | 100 | Unconditional Branch |
| BSA | 101 | Branch and save address |
| ISZ | 110 | Inc. and Skip if Zero |

# Module2 Background

- **Study of a Simple Machine**
  - **Register reference instructions**: These instructions explicitly refer to a register as the operand.
  - Bit pattern in operand part determines the register involved and the exact nature of the operation.
  - 12-instructions are in this group:

    CLA (7800), CIR )40),

    SZA (7004), CM 7100), SPA (7010), HLT (70



# Module2 Background

- **Study of**

| Symbol | Code (Hex) | Description |
|---|---|---|
| CLA | 7800 | Clear Ac |
| CLE | 7400 | Clear E |
| CMA | 7200 | Complement AC |
| CME | 7100 | Complement E |
| CIR | 7080 | Circular shift right AC |
| CIL | 7040 | Circular shift left AC |
| INC | 7020 | Increment AC |
| SPA | 7010 | Skip if Positive AC |
| SNA | 7008 | Skip if Negative AC |
| SZA | 7004 | Skip if Zero AC |
| SZE | 7002 | Skip if Zero E |
| HLT | 7001 | Halt |

# Module2 Background

- **Study of a**

| Symbol | **Semantic** |
|---|---|
| CLA | $AC \leftarrow 0$ |
| CLE | $E \leftarrow 0$ |
| CMA | $AC \leftarrow \overline{AC}$ |
| CME | $E \leftarrow \overline{E}$ |
| CIR | $AC \leftarrow shr\ AC, AC(15) \leftarrow E, E \leftarrow AC(0)$ |
| CIL | $AC \leftarrow shl\ AC, AC(0) \leftarrow E, E \leftarrow AC(15)$ |
| INC | $AC \leftarrow AC + 1$ |
| SPA | If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$ |
| SNA | If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$ |
| SZA | If $(AC = 0)$ then $(PC \leftarrow PC + 1)$ |
| SZE | If $(E = 0)$ then $(PC \leftarrow PC + 1)$ |
| HLT | $S \leftarrow 0$ (S is a start-Stop flip flop) |

# Module2 Background

- **Study of a Simple Machine**
  - **Input-Output instructions**: Instructions in this group are used to carry out the I/O operations.
  - Bit pattern in operand part determines the nature of I/O operation, i.e., type of I/O operation, type of I/O device, and

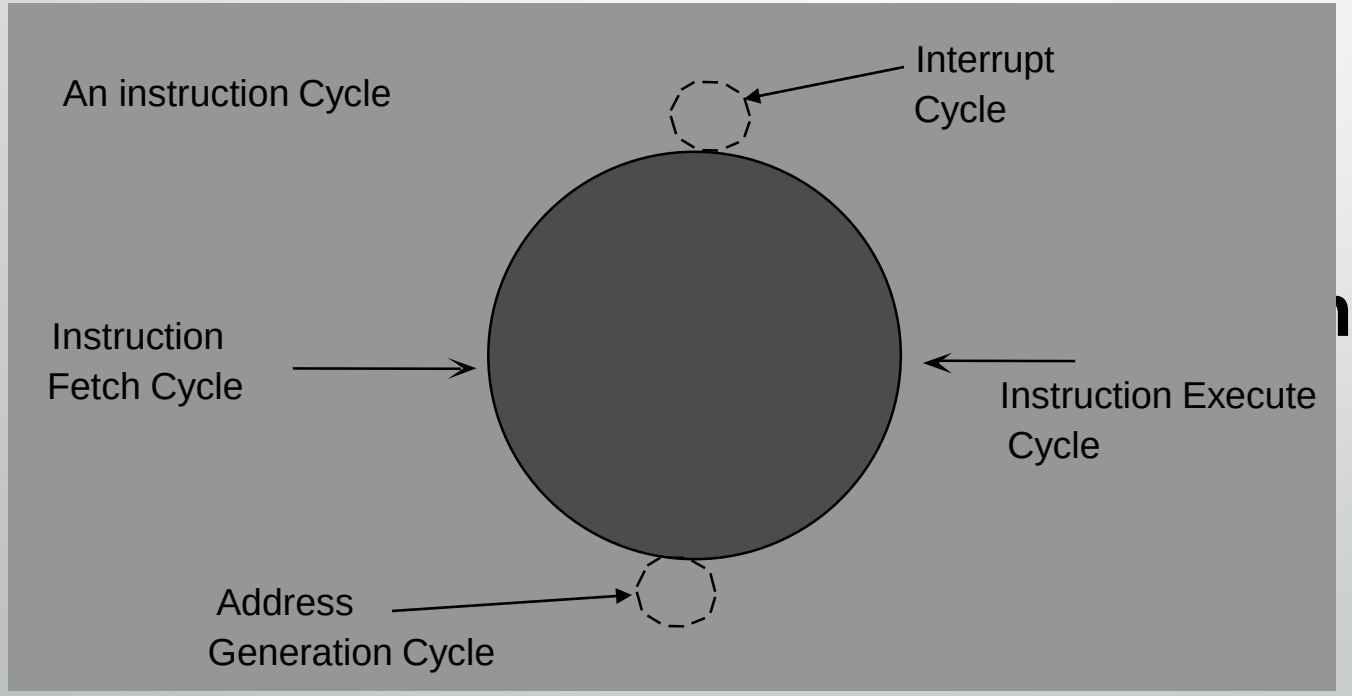
# Module2 Background

- **Study of a Simple Machine**
  - In general, the instruction set of a simple machine covers the following classes of operations:
    - Arithmetic, Logic, and Shift operations.
    - Memory access operations.
    - Control transfer operations.
    - I/O operations.
    - Start/Stop operations.

# Module2 Background

- **Study of a Simple Machine – Flow of Control and Data**
  - 1) Program and data are read in and stored in main memory.
  - 2) An instruction is fetched from main memory.
  - 3) Fetched instruction is decoded and investigated.
  - 4) Operand(s) is(are) fetched.
  - 5) Operation is performed.
  - 6) Steps 2-5 are repeated up to the end of program.
  - Steps 2-5 are called the **instruction cycle**.
  - Program counter is used to fetch an instruction.
  - Usually steps 2 and 3 are called **instruction fetch cycle** and steps 4 and 5 are called **instruction execute cycle**.
  - In order to execute a program, the system goes into a sequence of instruction cycles. Within each cycle an instruction is fetched, decoded, and executed.

# Module2 Background



# Module2 Background

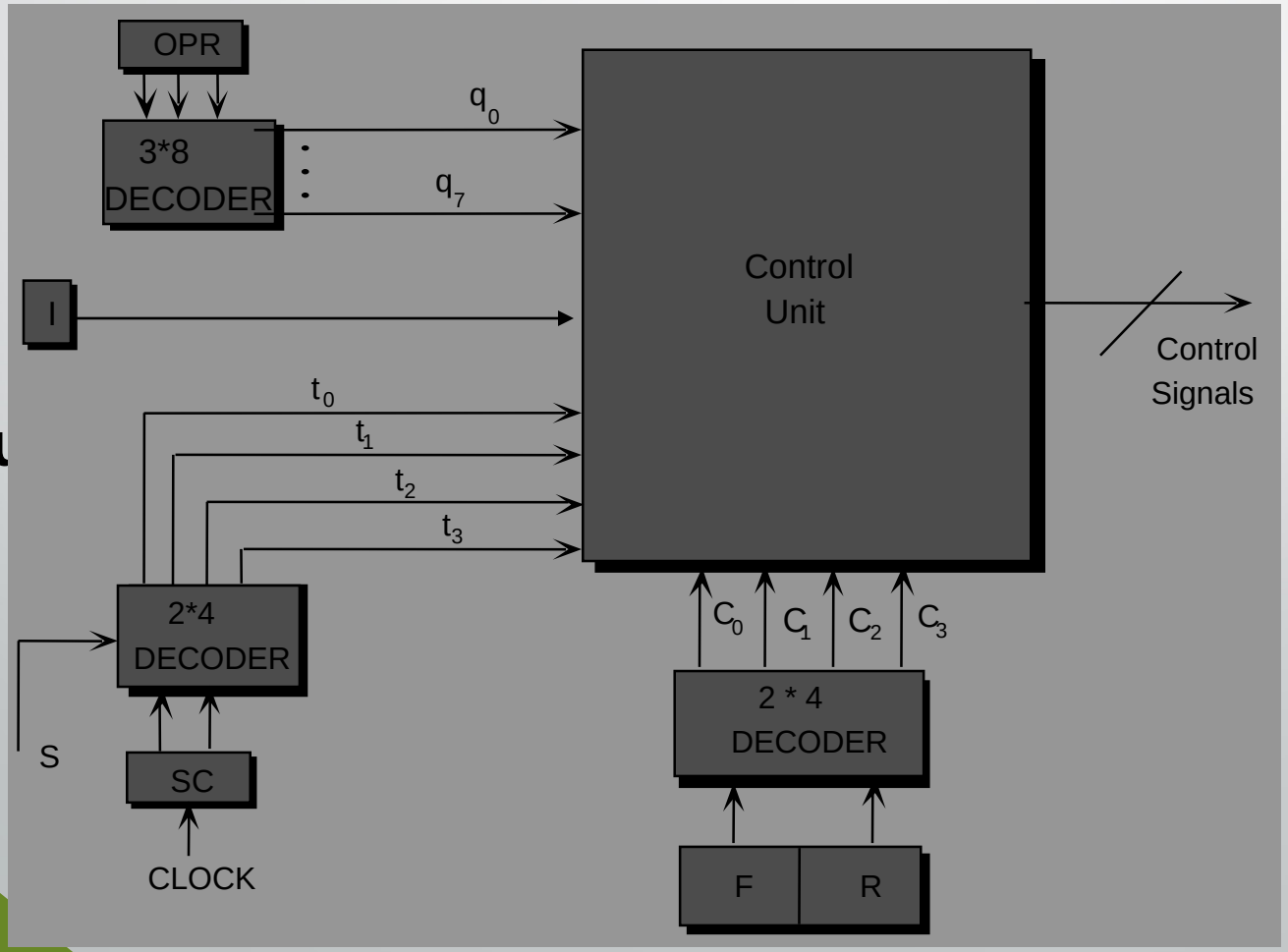
- **Study of a Simple Machine**
  - At each moment the system should know exactly which cycle it is in. In the simple machine this will be accomplished by the contents of 2 Flip Flops, called F, R Flip Flops.

| F | R | Nmonic |
|---|---|---|
| 0 | 0 | $C_0$ (Fetch cycle) |
| 0 | 1 | $C_1$ (Indirect cycle) |
| 1 | 0 | $C_2$ (Execute cycle) |
| 1 | 1 | $C_3$ (Interrupt cycle) |

  - Each cycle is a collection of several μ-operations. The μ-operation in each cycle should be executed in an orderly fashion. This order is enforced by means of a clock. In our simple machine each cycle contains four clock pulses.

# Module2 Background

- **Stu**

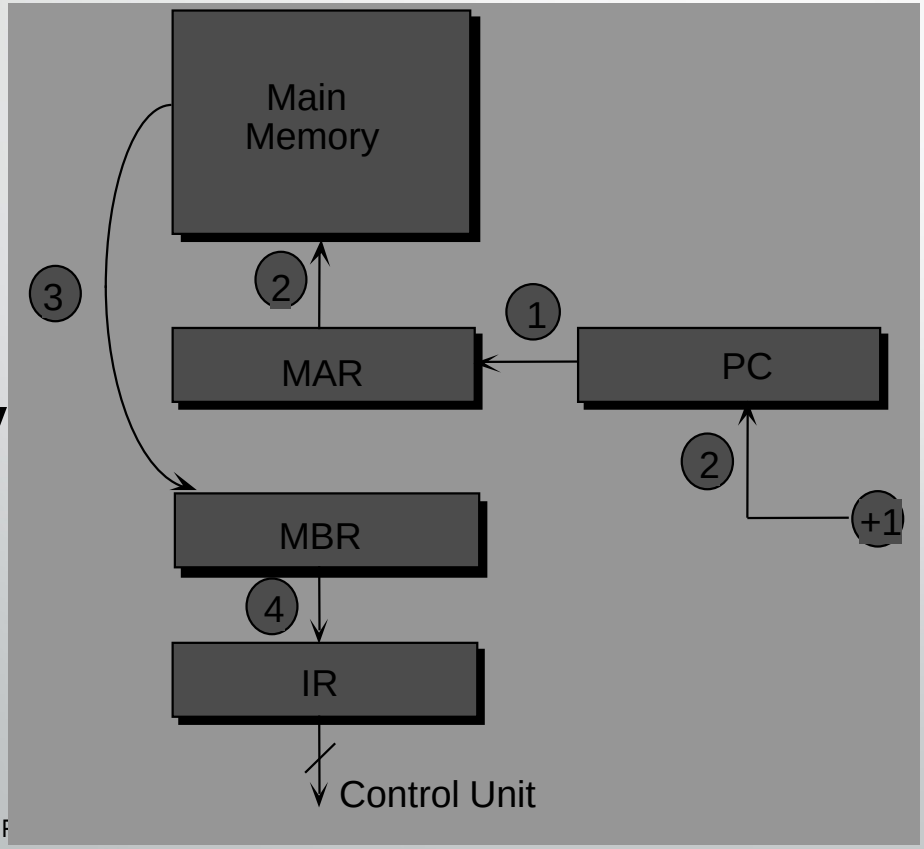


# Module2 Background

- **Study of a Simple Machine**
  - **Fetch Cycle**
    - Program counter is used to fetch an instruction from main memory.
    - Address of the next instruction is calculated.
    - The fetched instruction is interpreted.
      - MAR ← address of next instruction (PC)
      - MBR ← (M[MAR])
      - IR ← (MBR)
      - PC ← (PC) + 1* (The instruction length is assumed to be the same as the word length).

# Module2 Background

- **Study**



# Module2 Background

- **Study of a Simple Machine**
  - **Fetch Cycle**

$c_0t_0$: MAR $\leftarrow$ (PC)

$c_0t_1$: MBR $\leftarrow$ (M[MAR]), PC $\leftarrow$ (PC) + 1

$c_0t_2$: OPR $\leftarrow$ (MBR(op)), I $\leftarrow$ (MBR(I))

$q'_7Ic_0t_3$: R $\leftarrow$ 1

$(q_7+I')c_0t_3$: F $\leftarrow$ 1

# Module2 Background

- **Study of a Simple Machine**
  - **Indirect cycle**

$c_1t_0$: MAR $\leftarrow$ (MBR(ADDRESS))

$c_1t_1$: MBR $\leftarrow$ (M[MAR])

$c_1t_2$:idle

$c_1t_3$: F $\leftarrow$ 1, R $\leftarrow$ 0

# Module2 Background

- **Study of a Simple Machine – Execute cycle**
  - **AND A**

    $q_0c_2t_0$: MAR ← (MBR(address))
    $q_0c_2t_1$: MBR ← (M[MAR])
    $q_0c_2t_2$: AC ← (AC) Λ (MBR)
    $q_0c_2t_3$: ...

  - **ADD A**

    $q_1c_2t_0$: MAR ← (MBR(address))
    $q_1c_2t_1$: MBR ← (M[MAR])
    $q_1c_2t_2$: AC ← (AC) + (MBR)
    $q_1c_2t_3$: ...

# Module2 Background

- **Study of a Simple Machine — Execute cycle**
  - **LDA A**

    $q_2c_2t_0$: MAR ← (MBR(address))
    $q_2c_2t_1$: MBR ← (M[MAR])
    $q_2c_2t_2$: AC ← (MBR)
    $q_2c_2t_3$: ...

  - **STA A**

    $q_3c_2t_0$ : MAR ← (MBR(address))
    $q_3c_2t_1$ : MBR ← (AC)
    $q_3c_2t_1$ : M[MAR] ← (MBR)
    $q_3c_2t_1$ : ...

# Module2 Background

- **Study of a Simple Machine — Execute cycle**
  - **BUN A**

    $q_4c_2t_0$ : PC ← (MBR(address))

    $q_4c_2t_1$ : idle

    $q_4c_2t_2$ : idle

    $q_4c_2t_3$ : ...

  - **ISZ A**

    $q_5c_2t_0$ : MAR ← (MBR(address))

    $q_5c_2t_1$ : MBR ← (M[MAR])

    $q_5c_2t_2$ : MBR ← (MBR) + 1

    $q_5c_2t_3$ : M[MAR] ← (MBR), if (MBR=0) then PC ← (PC)+1

# Module2 Background

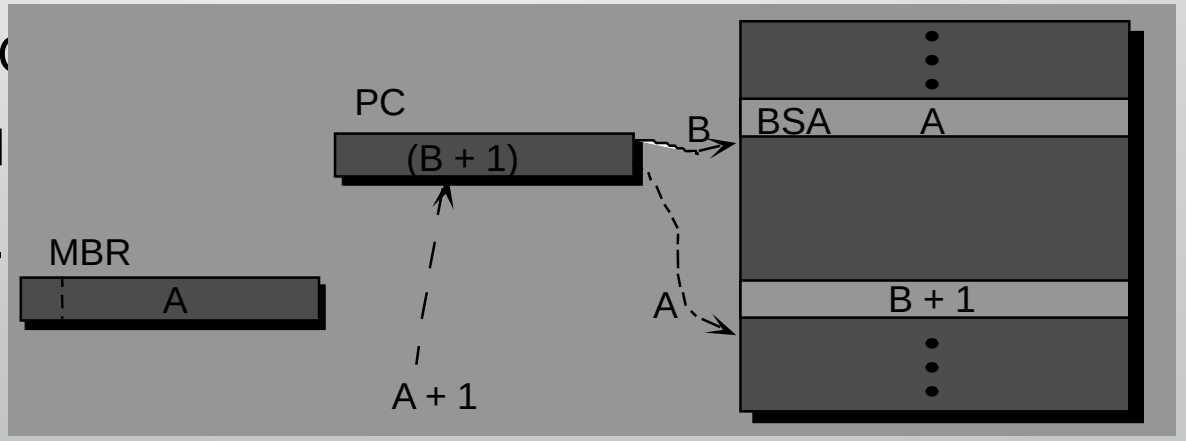
- **Study of a Simple Machine — Execute cycle**
  - **BSA A**

$q_6c_2t_0$: MAR ← (MBR(address)), MBR(address) ← (PC)

$q_6c_2t_1$: PC

$q_6c_2t_2$: M

$q_6c_2t_3$: ...



# Module2 Background

- **Study of a Simple Machine – Execute cycle**
  - **Register reference instructions**: Format of execute cycle for instructions in this class is almost uniform:
    - CLA:

      $q_7 I' c_2 t_0 (MBR_5)$: AC ← 0

      $q_7 I' c_2 t_1 (MBR_5)$: idle

      $q_7 I' c_2 t_2 (MBR_5)$: idle

      $q_7 I' c_2 t_3 (MBR_5)$: ...

    - CLE:

      $q_7 I' c_2 t_0 (MBR_6)$: E ← 0

      $q_7 I' c_2 t_1 (MBR_6)$: idle

      $q_7 I' c_2 t_2 (MBR_6)$: idle

      $q_7 I' c_2 t_3 (MBR_6)$: ...

# Module2 Background

- **Study of a Simple Machine — Execute cycle**
  - The uniform execute cycles for instructions in this class suggest a simplified control unit.
  - Practically every register reference instruction can be executed in one clock pulse.

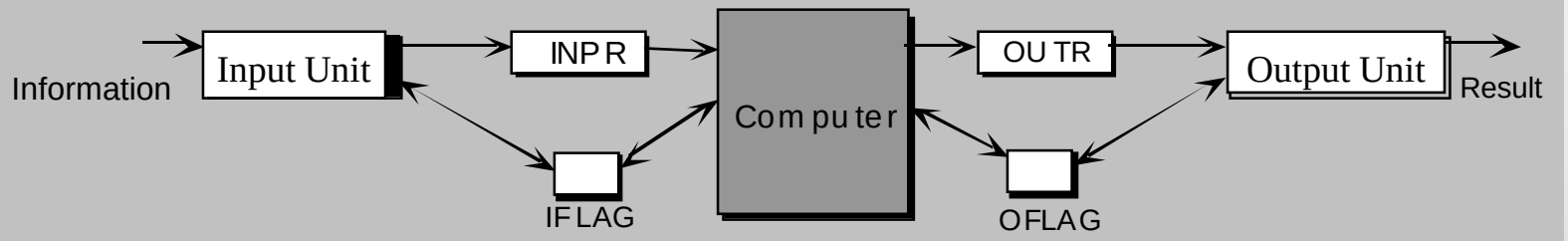
# Module2 Background

- **Study of a Simple Machine** – I/O Operations
  - A computer can serve no useful purpose unless it communicates with the outside world. Instructions and data are received via input devices and results are transferred to the outside world through output devices.
  - I/O devices are either mainly mechanical devices or partially mechanical devices.
  - It is known that mechanical devices are slower than electronic devices. This suggests that I/O devices (I/O operations) are much slower than the CPU and main memory (central) operations.

# Module2 Background

- **Study of a Simple Machine** – I/O Operations
  - Whenever several devices with different speeds are cooperating and/or competing over resources two major problems could arise:
    - Data could be lost.
    - Efficiency of faster units could be compromised.
  - Therefore to preserve the integrity and efficiency, one has to devise proper mechanisms to:
    - Synchronize I/O operations with the operations in main frame.
    - Utilize resources, i.e., CPU and main memory, efficiently.

# Module2 Background




Initially: IFLAG = 0

IFLAG = 0 ⇒ New data can be Read into INPR

IFLAG = 1 ⇒ New data cannot be Read in

Initially: OFLAG = 1

OFLAG = 1 ⇒ OUTR can be Written to

OFLAG = 0 ⇒ New Data in OUTR

# Module2 Background

**Input device:**
AA. If IFLAG =1 then goto AA.
INPR ← "data", IFLAG←1.

**Computer:**
BB. If IFLAG =1 then PC←PC+1,
Goto BB.
AC←INPR, IFLAG←0.

**Output Device:**
CC. If OFLAG =1 then goto CC.
"data" ← OUTR, OFLAG←1.

**Computer:**
DD. If OFLAG =1 then
PC←PC+1,
Goto DD.
OUTR←AC, OFLAG←0.

# Module2 Background

- **Study of a Simple Machine** – I/O Operations
  - Let us introduce an interrupt-bit (INT). At the end of each execute cycle, the contents of INT is investigated. If INT = 0, CPU continues its normal sequence of operations. If INT = 1, then control transfers to a specific procedure (Interrupt Handling Program).
  - In **Interrupt Handling Program**, the proper actions needed to resolve the interrupt will be performed.

# Module2 Background

- **Study of a Simple Machine** – I/O Operations
  - In simple machine at the end of each execute cycle - i.e., $c_2q_it_3$ ($0 \le i \le 7$), we have:

    $c_2q_it_3$: If INT $\Lambda$ (IFLAG V OFLAG) = 1 then R $\leftarrow$ 1,

    If [INT $\Lambda$ (IFLAG V OFLAG)=0] then F $\leftarrow$ 0

  - In the interrupt cycle:

    $c_3t_0$: MBR(address) $\leftarrow$ (PC), PC $\leftarrow$ 0

    $c_3t_1$: MAR $\leftarrow$ 0, PC $\leftarrow$ (PC) + 1

    $c_3t_2$: M[MAR] $\leftarrow$ (MBR), INT $\leftarrow$ 0

    $c_3t_3$: F $\leftarrow$ 0, R $\leftarrow$ 0

# Module2 Background

- **Study of a Simple Machine** — Interrupt
  - An interrupt mechanism is a built-in facility.  It is a signal to the processor that causes the processor, after competing its current instruction, to fetch the next instruction from a special location in memory.
  - The occurrence of an interrupt allows the processor to finish what it is currently doing with its program and then simply leave that program and begin executing another one.

# Module2 Background

- **Study of a Simple Machine** — Interrupt
  - The concept of program interrupt is used to handle a variety of problems which arise out of the normal program sequence.
  - Program interrupt refers to the transfer of control from the currently running program to another service program as a result of an **external** or **internal** generated request. Control might return to the original program after the service program is executed.

# Module2 Background

- **Study of a Simple Machine** — Interrupt
  - **Types of Interrupt**
    - **External Interrupt** is initiated mainly by external devices, such as I/O devices, to the main frame.
    - **Internal Interrupt** is initiated as a result of executing an instruction.
    - **Software Interrupt** is initiated by executing an instruction. It is a special instruction call that behaves like an interrupt rather than a subroutine call. It can be initiated by the programmer at any time in the program.

# Module2 Background

- **Study of a Simple Machine** – Interrupt
  - Internal Interrupt (Examples):
    - Arithmetic results out of range
    - Parity error
    - Illegal operation code
    - Stack overflow/underflow
    - Out of memory access
  - External Interrupt (Examples):
    - Input/output devices started earlier is finished
    - Parity error
    - Another processor needs attention
    - Operator at console

# Module2 Background

- **Questions**
  - What is the application of BSA?
  - In the simple machine, what is the last instruction of a subroutine?
  - How does the interrupt facility improve the efficiency of I/O operations?
  - Interrupt vector?
  - Interrupt within interrupt?
  - Format of interrupt handling program?

# Module2 Background

- **Programming Level**
  - Addressing Mode: The way operand is defined in the instruction.
  - Effective address: The address of the operand value.
  - Scratch pad memory: A collection of the resisters organized and accessed like the main memory words.
  - Base addressing: This is similar to the index addressing. The contents of a register (base register) will be added to every addresses in a program. Before the execution of the program the contents of the base register will be loaded. This allows the program to be loaded anywhere in the main memory. The contents of the base register should be gaurded during the course of the program execution.

# Module2 Background

- **Programming Level**
  - Relative addressing mode: The effective address is defined relative to the current contents of the program counter.
  - Instruction set: The collection of instruction supported by the machine.
  - Instruction cycle: The sequence of microoperations needed to fetch and execute and instruction.
  - Instruction fetch cycle: A subset of the instruction cycle to allow to fetch the next machine instruction.
  - Machine instruction: Binary representation of the assembly instruction.

# Module2 Background

- **Programming Level**
  - Instruction execute cycle: A subset of the instruction cycle that facilitates the execution of a machine instruction.
  - Address generation cycle: A subset of the instruction cycle that calculates the effective address of the operand.
  - Interrupt Handling Program: A collection of system routines developed for handling different variation of interrupts.

# Module2 Background

- **Programming Level**
  - Precise interrupt: During the course of the execution of the program, the location of interrupt can be determined.
  - Imprecise interrupt: During the course of the execution of the program the approximate position of the interrupt can be determined.
  - Extended opcode: A technique that allows to extend the instruction set by using part of the operand section of the instruction as a modifier to the op-code.

# Module2 Background

- **Programming Level**
  - Basic Block: A sequence of sequential instructions, i.e., a sequence of instructions without any branches (except possibly the last instruction).
  - Program Counter: A special purpose register that holds the address of the next instruction in sequence.
  - Stack: A form of data structure that supports last-in-first-out policy.

# Module2 Background

- **Questions**
  - Rewrite the indirect cycle to allow multilevel of indirections.
  - Within the scope of the simple machine, it is possible to increase the set of the register reference instructions?
  - The length of the opcode for the simple machine is 3. So by a simple calculation, the simple machine should have an instruction set of size 8! However, as we have seen, the simple machine has an instruction set of size 31! Explain.
  - What is the application of BSA?

# Module2 Background

- **Questions**
  - Does the simple machine support a recursive call?
  - What is the application of ISZ?
  - Within the scope of the simple machine, what is the last instruction of a subroutine?
  - For the simple machine, write an assembly program to perform A - B?
  - Does the synchronization scheme, discussed for the I/O operation, improve computer utilization?
  - How does the interrupt facility improve the efficiency of I/O operation?

# Module2 Background

- **Questions**
  - What is a “subroutine”?
  - What is a “subroutine call”?
  - What is the difference between a subroutine call and a goto statement?
  - What is a “Jump-and Link” Instruction?